PLAYING SOUND LISTS WITH *THE MISSING LINK*

The XB Game Developer's Package has two sound list compilers that can convert CALL SOUND statements into sound lists. The sound lists are loaded into VDP memory, where the custom music player in XB256 can play them. Once started, they play automatically, freeing your XB or compiled program to do other things.

It would be a powerful addition to The Missing Link if it could could play sound lists the same way. The problem is that almost all of the VDP memory is used for the bit mapped graphics. In theory it would be possible to play small sound lists from VDP memory, but it would not be very practical.

Another possibility would be to put the sound list in the 32K expansion memory. The 8K low memory is not an option because all of it is used by *The Missing Link* code. However, high memory has 24K of ram. With some clever programming tricks, it is possible to embed a custom sound list player and sound lists into high memory along with an XB program. Here's how.

FOR THE MISSING LINK RUNNING FROM XB

The first step is to make the sound lists. This process is discussed in the XB256 manual in pages 12-21. You should study that material to for a better understanding of sound lists. To get you started, here is a brief description of the process. The latest XBGDP package, JUWEL7, should be in disk 1.

First you write an XB program that plays the music and sound effects. The example program was:

```
100 !CHORD !entry point at start of first sound list
110 FOR I=1 TO 2
120 CALL SOUND(600,233,0):: CALL SOUND(600,233,0,294,0)
125 CALL SOUND(600,233,0,294,0,349,0):: CALL SOUND(250,233,30)
130 NEXT I !When done looping continue to line 140
140 !SCALE !alternate entry point. Line 210 loops back to here.
150 CALL SOUND(600,233,0):: CALL SOUND(600,262,0):: CALL SOUND(600,294,0)
155 CALL SOUND(600,311,0):: CALL SOUND(600,349,0)
160 CALL SOUND(600,392,0):: CALL SOUND(600,440,0)
180 CALL SOUND(600,466,0):: CALL SOUND(600,440,0):: CALL SOUND(600,392,0)
190 CALL SOUND(600,349,0):: CALL SOUND(600,311,0):: CALL SOUND(600,294,0)
195 CALL SOUND(600,262,0):: CALL SOUND(1200,233,0)
200 CALL SOUND(600,233,30)
210 GOTO 140 !endless loop to line 140
220 STOP !turn off all sound generators
230 !CHARGE !entry point at start of second sound list
240 !PLAYER2 !tells the compiler to use player2
250 CALL SOUND(100,466,0):: CALL SOUND(100,587,0):: CALL SOUND(100,698,0)
255 CALL SOUND(225,932,0):: CALL SOUND(75,784,0):: CALL SOUND(600,932,0)
260 STOP !turn off all sound generators
```

You can test the sounds with RUN 100, RUN 140, and RUN 230. Then save the file.
SAVE DSK2.SCALE
SAVE DSK2.SCALE-M,MERGE   save s the program in merge format.

Now we create the sound table.

RUN "DSK1.SLCOMPILER" to run the sound list compiler from the XBGDP disk.
When it runs, you get two prompts:

XB Sound program to load?	enter DSK2.SCALE-M
Sound List Output File?	enter DSK2.SCALE.TXT

Press Enter.and a sound list is created that runs from VDP ram and is compatible with XB256.

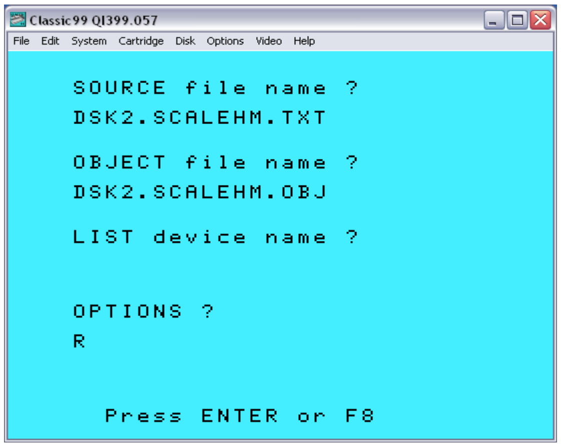
But the sound list player for TML reads the sound list from cpu memory, not vdp memory, so a small change is necessary. Load the file with a text editor and delete the two lines with AORG and SLEND.

```
       DEF CHORD
       DEF SCALE
       DEF CHARGE

       AORG >A000            Delete this line
       DATA SLEND,0,0        Delete this line
CHORD
       sound list continues......
```

Save the modified file as SCALEHX.TXT.. To avoid confusion, HX was appended to identify the file as one that runs in high memory under XB.

Now the file must be assembled. Go back to the XBGDP main menu and select the assembler.
After entering the source file name and the object file name, the screen should look like this:



```
SOURCE file name ?
DSK2.SCALEHM.TXT

OBJECT file name ?
DSK2.SCALEHM.OBJ

LIST device name ?


OPTIONS ?
R


  Press ENTER or F8
```

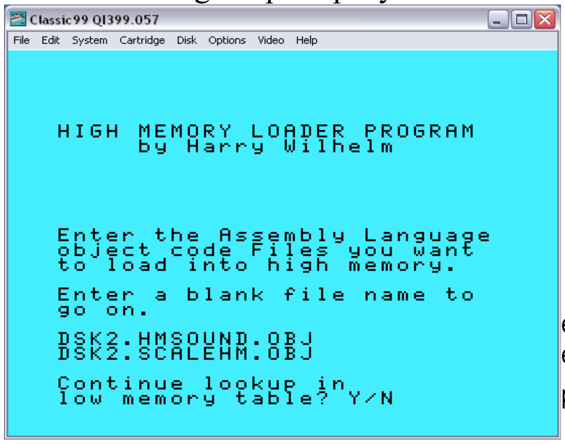
Press Enter to compile the sound list into object code compatible with the high memory player.

Now that the sound list has been assembled, we can combine it with a Missing Link program.

HMSOUND.OBJ is the sound list player for high memory. It contains an assembly subroutine accessed with CALL LINK("PLAYHM","NAME"). NAME is the name of the sound list to play. In the example above it would be CHORD, SCALE or CHARGE.

We will embed PLAYHM and SCALE in an XB program.
From the XB command line, (not running TML) enter: RUN "DSK1.HMLOADER"

After answering the prompts you see:



```
HIGH MEMORY LOADER PROGRAM
     by Harry Wilhelm


Enter the Assembly Language
object code files you want
to load into high memory.
Enter a blank file name to
go on.
DSK2.HMSOUND.OBJ
DSK2.SCALEHM.OBJ
Continue lookup in
low memory table? Y/N
```

enter DSK2.HMSOUND.OBJ
enter DSK2.SCALE.OBJ
press Y to continue lookup in low memory table, making TML available.

Press Y and an XB program is created containing the embedded code for HMSOUND and SCALEHX.

Save this program as DSK2.SCALE1. Now load *The Missing Link,*. then OLD DSK2.SCALE1
(With XB 2.9 you can simply CALL TML and the program is still loaded.)

With TML loaded and SCALE1 in memory, add this program to test it out. The lines can be typed in, merged, or added with Paste XB

```
3 CALL LOAD(8192,255,178)           !this is put there by HMLOADER
10 CALL LINK("CLEAR")
11 ANG=90+RND*180 :: CALL LINK("PUTPEN",92,120,0):: FOR I=1 TO 120 :: CALL
LINK("FWD",I,ANG)
20 CALL KEY(0,K,S):: IF S<1 THEN 50
25 IF K=65 THEN CALL LINK("PLAYHM","CHORD"):: GOTO 50
30 IF K=66 THEN CALL LINK("PLAYHM","SCALE"):: GOTO 50
35 IF K=67 THEN CALL LINK("PLAYHM","CHARGE"):: GOTO 50
50 NEXT I :: GOTO 10
```

This program draws polyspirals. After each line is drawn, it checks to see if a key is pressed. If A is pressed it plays CHORD. If B is pressed it plays SCALE. If C is pressed it plays CHARGE.

You can load more than one sound table using HMLOADER. Just be sure to delete the lines with AORG and SLEND.

Although the focus here is to play sound lists with *The Missing Link*, HMSOUND does not have to be run under TML. T40XB, T80XB, and even plain XB programs can use it as well.

FOR THE MISSING LINK RUNNING FROM COMPILED CODE

Create the sound table just like you did above.
RUN “DSK1.SLCOMPILER”
Enter DSK2.SCALE-M”
Enter DSK1.SCALE.TXT”   and the sound table SCALE.TXT is created.

Now SCALE.TXT must be modified for compatibility with the XB compiler.
Open SCALE.TXT with a text editor such as Notepad.
Delete the two lines with AORG and SLEND.
Delete the last line with END

Now you must recreate the DEF table at the top of the sound table

```
       DEF CHORD     \
       DEF SCALE       delete these lines
       DEF CHARGE    /
```

And replace them with:

```
DTSTRT TEXT 'CHORD '      For the assembler, strings are quoted with apostrophes.
       DATA CHORD
       TEXT 'SCALE '      The strings must be 6 characters long. Add spaces if necessary.
       DATA SCALE
       TEXT 'CHARGE'
       DATA CHARGE
DTEND
       sound list continues....
```

Save the modified file as SCALEHC.TXT I
To avoid confusion, HC is appended to show the file is one that runs in high memory with the compiler.
This file does not have to be assembled. That will happen in the compilation process.

There are other changes to be made. The compiler must be modified to recognize the code we are adding. The compiler manual discusses this in detail on pages 22-25, and you should study that material to understand this better. To get you started, here is a quick description of the process.

OLD DSK1.COMPILER
Add these lines:
10000 DATA PLAYHM   Now the compiler will recognize PLAYHM as part of the runtime code.
10020 DATA DSK2.TMLSOUNDC.TXT   This will load the sound player for compiled TML.
Be sure lines 10000 and 10020 are each followed by a single null string. The DATA statements are shown below.

```
10000 DATA PLAYHM
10010 DATA ""
10020 DATA DSK2.TMLSOUNDC.TXT       or TMLSOUNDC.TXT if you are using Assm994a
10030 DATA ""
```

SAVE DSK1.COMPILER after making the changes.

Now the XB program must be modified. Add line 1 CALL LINK("TML16").
Then all the TML call links must be converted to lower case. This program is short enough that you could do it by hand, but usually it would be done with:
CALL LOAD("DSK1.UC2LC")
CALL LINK("X")   and the call links are converted to lower case.

One more step is needed. **A CALL LINK that runs natively in the compiler must be in upper case.** So you have to go through the program and convert all the CALL LINK("PLAYHM",...) back to upper case. In a long program, Find/Replace in the text editor is very helpful.

When you are done, the program should look like this:

```
1 CALL LINK("tml16")           !add this line to initialize TML
10 CALL LINK("clear")
11 ANG=90+RND*180 :: CALL LINK("putpen",92,120,0):: FOR I=1 TO 120 :: CALL
LINK("fwd",I,ANG)
20 CALL KEY(0,K,S):: IF S<1 THEN 50
25 IF K=65 THEN CALL LINK("PLAYHM","CHORD"):: GOTO 50
30 IF K=66 THEN CALL LINK("PLAYHM","SCALE"):: GOTO 50
35 IF K=67 THEN CALL LINK("PLAYHM","CHARGE"):: GOTO 50
50 NEXT I :: GOTO 10
```

Save this program as TMLPOLYL and TMLPOLYL-M,MERGE
The "L" was appended to show that the CALL LINKs have been converted to lower case.

At last it is time to compile TMLPOLYL-M.

The program is compiled in the normal way.
Be sure to choose N for Runtime in low mem? (because TMLC has to be in low memory.)
When the compiler comes to "Add more runtime routines?," enter Y..
DSK2.TMLSOUNDC.TXT is suggested. (you added this in line 10020) Press Enter.
DSK2.FILE.TXT is suggested. Change this to the name of your sound list, DSK2.SCALEHC.TXT.
Press Enter.
There are no more runtime routines to add, so at the next prompt press Fctn 3 to clear the line, then press Enter.

Assemble the file as usual.

At the Loader, when asked “using assembly support?”, enter Y.
For “Compiled file to load”   enter DSK2.TMLPOLYL.OBJ
For “Assembly routines to load”   enter DSK1.TMLC
Then keep pressing enter until finished.

Now you can enter RUN to see how the program works.

Nothing more needs to be done to run the program in EA5 format.

If you want to run the program in XB, it needs to be in 2 parts. One will be loaded into low memory, and the other into high memory. You just created the high memory part when you saved the file TMLPOLYL-X. But it will not run unless the assembly routines contained in TMLC are first loaded into low memory. Here is how to do this.

Enter OLD DSK1.TMLC, then add the highlited code to line 10:

```
10 CALL INIT :: CALL LOAD(8192,255,172):: CALL LINK("X"):: RUN "DSK2.TMLPOLYL-X"
```

Then SAVE DSK2.TMLPOLYL-Y

Now when you RUN “DSK2.TMLPOLYL-Y” it will first load the TML support routines into low memory, then load and run TMLPOLYL-X, which is the high memory part of the compiled code.

Some of the code for the sound player is built into TMLC, Because of this, .TMLSOUNDC will only work with compiled TML programs.

If you want to use sound lists compiled programs using T40XB, T80XB, or other XB programs you must use HMSOUNDC.TXT instead of TMLSOUNDC.

COMPRESSION UTILITIES FOR *THE MISSING LINK*

TMLCOMP is a utility written specifically for *The Missing Link*. It lets you select sprite attributes and patterns in the VDP ram to save as MERGE format disk files containing compressed images of the sprite data. When these files are merged into an XB program, the assembly subroutine WRITEC can be used to instantly restore the sprite data. Being able to restore this data quickly can be very useful when animating sprites. TMLCOMP and WRITEC are adapted from COMPRESS and CWRITE which are part of XB256. More information about these last two can be found in the XB256 manual.

TMLCOMP contains a subprogram called COMPRESS that is merged into your program. In a subprogram, all variables are local, and cannot conflict with variables in your program that may have the same name.

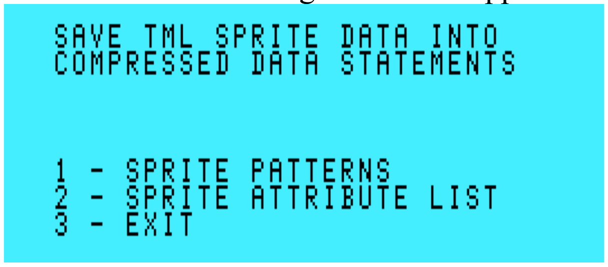
Following is a short example of how to use COMPRESS. Let's say you are writing a game for *The Missing Link* that uses sprites.

Create lines of code that define the sprite patterns with CALL LINK("CHAR",...) and create the sprites with CALL LINK("SPRITE",....). This can be part of the game program or a stand alone program. As always, test out the program to be sure the results are as expected.

For this example, let's say that lines 100-200 define sprites 4-12, then lines 300-400 gives those sprites new definitions and patterns.

When you are happy with the sprites, with the program loaded but not running, type:
MERGE DSK1.TMLCOMP, then add these lines to be executed after the two sets of sprites are created.
205 CALL COMPRESS
405 CALL COMPRESS

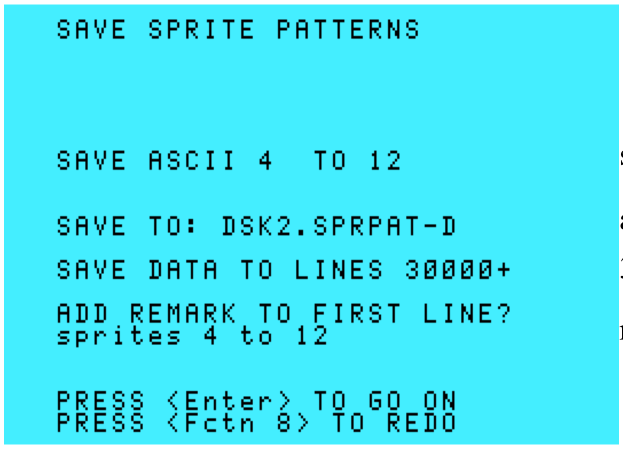
Run the program. The first sprites have been defined when it gets to line 205, then CALL COMPRESS is executed. The following screen will appear:



```
SAVE TML SPRITE DATA INTO
COMPRESSED DATA STATEMENTS


1 - SPRITE PATTERNS
2 - SPRITE ATTRIBUTE LIST
3 - EXIT
```

Press 1 to get the screen for saving sprite patterns, which looks like this after answering the prompts,



```
SAVE SPRITE PATTERNS



SAVE ASCII 4  TO 12

SAVE TO: DSK2.SPRPAT-D

SAVE DATA TO LINES 30000+

ADD REMARK TO FIRST LINE?
sprites 4 to 12

PRESS <Enter> TO GO ON
PRESS <Fctn 8> TO REDO
```

save 8 sprite patterns starting at ASCII 4

append a D to identify the file as data.

30000 was suggested as the starting line number

remark helps you know of what the DATA statements do

Press Enter to save the sprite patterns.

The sprite attribute list contains the ASCII pattern, the color, the row, and the column for each sprite.
Save the sprite attribute list for sprites 4-12, just like above, but there is one additional option:
BLANK HIGHER SPRITES Y/N
This depends on your application. If you choose Y then any sprites with a higher number will be invisible.

Now choose 3 to exit, and the program returns from the sub COMPRESS to the lines that define the second set of sprites. Repeat the process to save the second set of sprite patterns and sprite attributes.

At this point you will have created 4 files with patterns and attributes for the two sets of sprites.

Type NEW, merge the 4 files, then list the program

```
LIST
30000 !sprites 4-12
30001 RESTORE 30003
30002 READ _$ :: IF _$="" THEN RETURN ELSE CALL LINK("CWRITE",_$):: GOTO 30002
30003 DATA ";#H)€e6ÇdÛd#(*…áî˜#¼F%*#éIÅèØ×am-ô##ó⊠Ø?m&#Ÿ-{ßPfµƒÑ##·#å„ù#NÑÙ*
30004 DATA ""
30005 !sprites 4-12
30006 RESTORE 30008
30007 READ _$ :: IF _$="" THEN RETURN ELSE CALL LINK("CWRITE",_$):: GOTO 30007
30008 DATA "<%)¡?Æ˜¯#9™YŸÔ”h0rŠ„##ÊŸYM#Ñ#@œÞœQCƒ·ÇÐ"
30009 DATA ""

30010 !2nd sprites 4-12
30011 RESTORE 30013
30012 READ _$ :: IF _$="" THEN RETURN ELSE CALL LINK("CWRITE",_$):: GOTO 30012
30013 DATA ";#HpÍ#,û#Š#Ð°ÂìÇù#P#áOs ¥g{é###ºâ9T¶ÿ…m,”´fÜ‘*-„¯•µÉoàìX#R9^#y#Ìe#·Ê¢ 5¹h.$"
30014 DATA ""
30015 !sprites2 4-12
30016 RESTORE 30018
30017 READ _$ :: IF _$="" THEN RETURN ELSE CALL LINK("CWRITE",_$):: GOTO 30017
30018 DATA "<%•×Ð”ø˜ð‘-[Ï«Ý Þ0'šo²,žá±ýÉ#?Õ‘yŠa7_*Ð"
30019 DATA ""
```

Looking at the lines above, we can see that COMPRESS creates files containing:
-An optional comment
-RESTORE to point to the first data statement
-A one line subroutine to copy the data statements into vdp using CWRITE.
-DATA statements, ending with a null string so the loader knows when to stop.

In the example above, you *could* load sprite1 patterns and attributes with:
100 GOSUB 30000
101 GOSUB 30005

But if you will always be loading both together, it is neater and quicker to remove the highlited lines. Then 100 GOSUB 30000 or GOSUB 30010 will load all the data statements, only stopping when it gets to the null string in line 30009 or 30019.

With the sprites saved in data statements we can get rid of all the lines that define the character patterns and sprites, and replace them with our DATA statements. Then simply add a line with GOSUB 30000 or GOSUB 30010 to load the sprite data.

The DATA statements that are created can contain bytes from 0 to 255. They can be LISTed and RESequenced, but ***XB will not let you edit them*** because they contain characters not recognized by the line editor.

CALL LINK ("WRITEC",compressed string[, . . .])

WRITEC writes a compressed string to VDP ram starting at the memory address embedded in the beginning of the string. Compressed strings are created with the TMLCOMP utility as descibed above. Up to 16 compressed strings can be written with one call to WRITEC. It is the same as CWRITE for XB256, but it has been renamed to avoid confusing the compiler.

WRITEC for The Missing Link running from Extended BASIC.

WRITEC is not included in The Missing Link, and because there is no room for it in low memory, it must be embedded in the XB program and run from high memory. Here's how:
RUN "DSK1.HMLOADER"
At the prompts, enter DSK2.TMLCWRITE.OBJ   (This is part of XBGDP)
If you are also using sound lists, the files for them should be included by adding DSK2.HMSOUND.OBJ and DSK2.SCALEHM.OBJ as described above in the section about sound lists.
When finished, enter a blank line
Then enter Y to continue lookup in low memory table. This is needed so the TML routines can be found.
First save the program, then load *The Missing Link.* Reload your program and now you can add lines of XB/TML code to it as desired. The lines can be typed in, merged, or added with Paste XB

WRITEC for The Missing link running from compiled code.

To use in a compiled program, WRITEC must be added to the compiler.
OLD DSK1.COMPILER
Be sure these lines are in the compiler.
10000 DATA WRITEC   Now the compiler will recognize WRITEC as part of the runtime code.
10020 DATA DSK2.TMLCWRITC.TXT   This will load WRITEC for compiled TML.
Be sure lines 10000 and 10020 are each followed by a single null string. The DATA statements are shown below.)

```
10000 DATA WRITEC
10010 DATA ""
10020 DATA DSK2.TMLWRITC.TXT        or TMLWRITC.TXT if you are using Assm994a
10030 DATA ""
```

The compiler comes with these already added, but check to make sure they are there and that they use your preferred disk number.

SAVE DSK1.COMPILER after making the changes.

Now your XB program must be modified. Add line 1 CALL LINK("TML16"). This will set the registors for bit mapped mode and initialize the vdp memory.

Then all the TML CALL LINKS must be converted to lower case. Although you can do this by hand, usually it would be done with:
CALL LOAD("DSK1.UC2LC")
CALL LINK("X")   and the call links are converted to lower case.

BUT, any CALL LINK that runs natively in the compiler must be in upper case. So you have to go through the program and convert all the CALL LINK("writec",...) back to upper case. In a long program, Find/Replace in the text editor is very helpful.

Now the program can be saved, compiled, assembled, and run. The process is the same as described in the sound list section above.